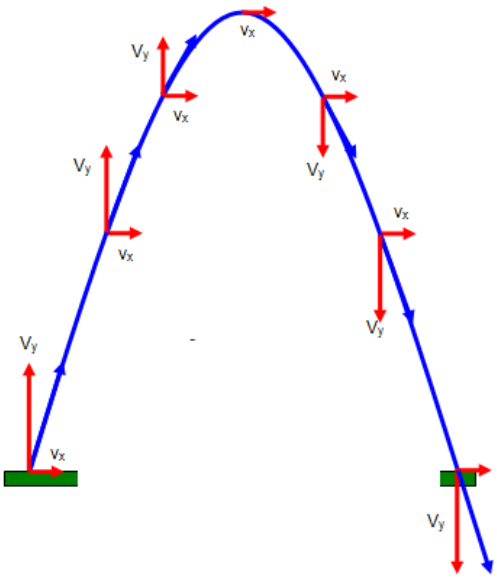
Physics based animation
Unit 01 - Introduction
Practical

Grégory Leplâtre

Semester 1 - 2017/2018



1 Introduction

The Goals of this first practical are to:

- Give you an opportunity to become familiar with the code that is provided to you.
- Write simple (time-based) animations
- Use the equations of motion to implement physically-plausible animations

Working with the code provided

A Visual Studio project and solution are already set up for you. The project includes the following library: `glew`, `glfw`, `glm` and `soil`. Note that `soil`, which is a useful library to work with images, isn't used by the project as no polygonal texturing functionality has been implemented.

To use the project:

1. Unzip `project1.zip`
2. Open the solution `Simulation.sln` (Using MS Visual Studio 2017)
3. Retarget the project: in the solution explorer, right-click on the solution and chosse `Retarget Projects`.

That's it, the project should now work on your system, which you can test: hit `ctrl-F5`.

2 Development approach

The focus of this module is the development of physics-based simulations, not graphics programming nor game engine development. Therefore it would seem reasonable to provide you with a game engine that contains everything you need except the physics part. However, I am keen for you to develop your software engineering skills by evolving the architecture of our application to meet our specific needs. These needs will evolve every week, so our architecture will be dynamic.

I also believe that you should be able to produce a realtime application using OpenGL from scractch *i.e.,* without relying on a third party framework. You will not start completely from nothing as I am providing you wil a small amount of code to save a bit of time an ensure that you can code animations in no time. Some of the code provided is unlikely to require any modification as it carries out basic tasks, such as dealing with shader files. Other classes will evolve with your project.

Let's examine the code given to you:

2.1 main

Rendering and interaction are handled by `main.cpp`. This is what the functions included in that file are for. As you will see, only the concepts of camera and mesh have been abstracted out of the main file. We'll look into these later, let's focus on the main by jumping straight to the `main()` function (at the bottom of the file).

The first call prepares for rendering. You are welcome to take a look at the implementation of the `initRenderer` method, which mostly relies on the GLFW library to do the job.

```
1  // init renderer
2  initRender();
```

The next bit demonstrates how to create a mesh and apply transformations to it. The Mesh class will be presented in more details later, but you can see that its use is very simple. You will notice that the default mesh constructor creates a quad.

```
// create ground plane
Mesh plane = Mesh::Mesh();
// scale it up x5
plane.scale(glm::vec3(5.0f, 5.0f, 5.0f));
```

Next, we create a "particle", which is simply a quad that has been rotated and scaled down.

```
// create particle
Mesh particle1 = Mesh::Mesh();
//scale it down (x.1), translate it up by 2.5 and rotate it by 90 degrees around the x axis
particle1.translate(glm::vec3(0.0f, 2.5f, 0.0f));
particle1.scale(glm::vec3(.1f, .1f, .1f));
particle1.rotate(M_PI_2, glm::vec3(1.0f, 0.0f, 0.0f));
// allocate shader
particle1.setShader(Shader("resources/shaders/core.vert", "resources/shaders/core_blue.frag"
  ));
```

Notice that `Mesh` objects have a `Shader` attribute. This will be discussed later.

Simple operations like these are all we need as far as geometry manipulation is concerned when developing physics simulation for particles. Only being able to work with quads would however be too limiting when we will tackle rigid body physics. The `Mesh` class allows one to create a mesh from a .obj file. Rendering improvements are obviously possible but unnecessary for the time being.

The comment that follows indicates where you should declare the particles that you will need to complete the tasks of this tutorial

```
/*
CREATE THE PARTICLE(S) YOU NEED TO COMPLETE THE TASKS HERE
*/
```

Next is the game loop, which contains all instructions that need to be executed at every frame. This is where simulations will be implemented or called, as well as where realtiem interaction and rendering takes place. Therefore, the code you will find in that loop pertains to the three following categories:

- Interaction (managing user actions)
- Animation/simulation
- Rendering

Animations are time-based, whether they are precomputed or simulated in real-time. GLFW gives us access to elapsed time in seconds, which we can use to keep track of frames in the game loop. This is a simple approach to timing that has limitations that we will discuss in a later unit.

```
GLfloat firstFrame = glfwGetTime();

// Game loop
while (!glfwWindowShouldClose(window))
{
  // Set frame time
  GLfloat currentFrame = glfwGetTime() - firstFrame;
  // the animation can be sped up or slowed down by multiplying currentFrame by a factor.
  currentFrame *= 1.5f;
  deltaTime = currentFrame - lastFrame;
  lastFrame = currentFrame;
```

The next part takes care of user interaction. The code is self-explanatory

```
    //*
    **  INTERACTION
    */

    // Check and call events
    glfwPollEvents();
    DoMovement();

    // view and projection matrices
    projection = glm::perspective(camera.GetZoom(), (GLfloat)SCREEN_WIDTH / (GLfloat)
    SCREEN_HEIGHT, 0.1f, 1000.0f);
    view = camera.GetViewMatrix();
```

We finally reach the place in the code where you will complete this week's tasks. Some of the tasks are one-liners, others will take a bit more effort.

```
    /*
    **  ANIMATIONS
    */

    // 1 - make particle fall at constant speed using the translate method


    // 2 - same as above using the setPos method


    // 3 - make particle oscillate above the ground plance


    // 4 - particle animation from initial velocity and acceleration


    // 5 - add collision with plane


    // 6 - Same as above but for a collection of particles
```

The final relevant part takes care of rendering every frame. All we care about is the method that allows us to render a mesh. This method called `draw` has been implemented in the main file. Make sure you add the appropriate draw calls on the particles you will create when you complete this practical's tasks. The snippet below contains two draw calls that draw the two objects created: the ground plane and the particle.

```
    /*
    **  RENDER
    */

    // Clear the colorbuffer
    glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // draw groud plane
    draw(plane);
    // draw particles
    draw(particle1);
```

2.2 Rendering and utilities

Object creation and manipulation

The `Mesh.h` and `Mesh.cpp` files define `Vertex` and `Mesh` classes that allows you to create geometry in two ways. You can use the default `Mesh` constructor, which creates a horizontal 1x1 quad centred on the origin. Alternatively, you can create a mesh from a .obj file. The `OBJLoader` class is responsible for loading .obj files.

The position, orientation and scale of a mesh are captured by its member attributes: `m_translate`, `m_rotate` and `m_scale`. These are standard 4x4 transformation matrices. There are get methods to access each of these transformation matrices, but only a set method has been implemented for the translate value (See `Mesh.h`). In fact, two set methods have been written for this purpose for convenience.

To manipulate a mesh, you can modify its translate, rotate or scale attribute directly (when a set method is implemented), or you can use the supplied `translate`, `rotate`, `scale` methods.

At this stage of the development, it has been decided that a mesh could be allocated a shader to control its look. This makes sense conceptually as the shader is only used to allocate a colour to the mesh. An alternative design could have involved defining a colour attribute for a `Mesh` object and hidden the concept of shader from the user. Some refactoring would definitely be necessary if we want to improve the look of our scenes, for example by adding illuminations and implementing illumination models such as Lambert or Blinn. For the moment, you only need to understand the logic behind the design and its limitations. For your information, shader files are handled by a `Shader` class. An object of that class is typically created from two attributes: a vertex shader file and a fragment shader file.

Camera

The camera code has been implemented in its own class. Note that all the code is contained in the header file. This class provides us with usual controls over the camera.

3 Tasks

For this first practical, you will not implement a realtime simulation in the strict term of the sense, but you will create animations based on the equations of motions covered in this unit. But before applying the equations of motion, let's start by simply making an object move. To give you an idea of what the outcome of each task should look like, refer to the relevant executables provided (task [i].exe).

Note that the way the tasks are designed reflects the way you should approach problem solving. If you were asked to complete Task 6, without any other instructions, a good approach would be to complete Task 1, then 2, etc and eventually Task 6.

3.1 Task 1 - particle movement

Task: Make a particle fall at constant speed using the translate method of the Mesh class.

This is to introduce you to working with time variables: `currentFrame` and `deltaTime` in the proposed implementation. The main GLFW function is `glfwGetTime()`.

3.2 Task 2 - particle movement variation

Task: Make a particle fall at constant speed using the setPos method of the Mesh class.

3.3 Task 3 - particle movement variation 2

Task: Make a particle oscillate above the ground plane.

Run the solution executable if you are unsure of the meaning of the question. This type of movement is typically achieved using a sine function.

3.4 Task 4 - particle trajectory from initial velocity and acceleration

Task: Animate a particle based on its initial position, velocity and acceleration (the latter being constant).

This is where you are supposed to apply the equations of motion. The three pieces of data will be 3D GLM vectors. Choose what you want for the initial position and velocity and use `a = (0, -9.8, 0)` for the acceleration.

3.5 Task 5 - Collision

Task: Make the particle bounce by adding a collision test with the ground plane.

Here's additional information to clarify the task:

- Assume that the ground plane is infinitely large, so testing for collision with the plane equates to testing that a particle is at the height of the plane (or close to it, or below).
- To implement a bounce, consider what needs to happen to the velocity when a collision is detected. Although we haven't covered the concept of energy loss/conservation, you can deal with the problem intuitively and consider different behaviours: a particle could lose energy on impact and see its velocity decrease, or it could bounce forever without ever stopping. You can also make it bounce with increased velocity any time there is contact with the plane. The first behaviour seems like a natural one for most physical surfaces, while the last one would make sense if the ground floor represents a pinball kicker.
- Add whatever variables you need.

Task: Same as Task 5, but for a collection of particles.

If you can get one particle to bounce around, you will undoubtedly want to see how many you can animate. To make the animation interesting, give them all different (random) initial velocities. There are two interesting aspects to this task:

1. Code change required to make it work, including data structures needed to hold information about the particles
2. Optimisation: how many particles can you animate concurrently? 1,000, 10,000, 1 million? I'm not asking you to make any change to the code that will increase this number (I'm not stopping you either!), but I'd like you to reflect on what could be done to improve performance.

4 Deliverables

There is no deliverable for this assignment.